The Ravenscar Profile for Real-Time and High Integrity Systems

Brian Dobbing
Praxis Critical Systems

Alan Burns
University of York

The Ravenscar Profile offers a unique opportunity to developers of real-time and high integrity systems. For the first time in the history of our industry, there is direct support for constructing deterministic, concurrent software within an international standard programming language. The Ravenscar Profile is founded on state-of-the-art, deterministic concurrency constructs defined in ISO standard Ada95. This results in a set of building blocks that are basic enough for constructing most types of real-time software, while also being sophisticated enough to minimize the risk of error associated with using low-level primitives such as not releasing a lock on all paths. These building blocks are also amenable to the many forms of analyses that can be applied during development to assure the correctness of complex real-time programs, including scheduling and response time analysis, data and information flow analysis, exception freedom, and formal analysis using theorem provers and model checkers. As a result, nonfunctional requirements such as timing and ordering constraints and resource utilization can be established early in the life cycle with consequent reductions in cost, delays, and risk of failure.

The Ravenscar Profile is now established as a state-of-the-art model for building safe and reliable real-time systems. The profile was originally defined in 1997 at a workshop of international real-time experts and is named after the village of Ravenscar in northern England where the workshop was held. It is specified as a subset of concurrency features in Ada95 that exhibit determinism in key areas such as timing, memory usage, and function behavior. The original definition has been slightly refined in light of the application experience, and the final definition is being incorporated into the next ISO standard revision of the Ada language [1].

Although the Ravenscar Profile is specified in Ada terms, it is based on a language-independent set of building blocks that are suitable for constructing typical real-time systems, and as input to analysis tools that provide evidence that the concurrency requirements of the system have been met.

Traditional methods of implementing concurrency in a predictable way have focused on approaches such as using cyclic executives that repeatedly execute a set of functions in a fixed order in preset time frames. However, such approaches have become inadequate as system complexity has increased and the burden of maintaining correct static timelines during system upgrade becomes prohibitive. This has led to wider acceptance of concurrent programming as the preferred approach.

Yet the quality of evidence for concurrency properties traditionally has been rather low. This is because guarantees such as sufficiency of scheduling to meet deadlines, accuracy in timing behavior, correct execution-time ordering of events, and correct levels of protection for access to shared data are difficult to establish for all possible operational scenarios by testing alone. In addition, the tools that may be used to provide this kind of evidence require specialized inputs that define deterministic timing behavior, often based on using a specific kind of model or restricted source language, and supported by a real-time operating system with totally deterministic timing characteristics.

> *"The advent of implementations of the Ravenscar Profile ... heralds the availability of the most rigorous environment for developing high-integrity concurrent programs."*

The advent of implementations of the Ravenscar Profile, including some with supporting evidence certifiable to standards such as Radio Technical Commission for Aeronautics, Inc. (RTCA) Defense Order (DO)-178B [2] level A, heralds the availability of the most rigorous environment for developing high-integrity concurrent programs. This offers a unique opportunity to developers of real-time and high integrity systems to be able to demonstrate early in the life cycle that nonfunctional requirements (such as failure modes, timing and ordering constraints, and predictable resource usage) are satisfied rather than discovering deficiencies during the integration phase when corrections are often very difficult and costly to implement.

In this article, we present the following: the motivation behind the creation of the Ravenscar Profile, a brief definition of its specification, the ways in which it may be used with verification tools to produce evidence of dependability, and a short concluding example.

Motivation

The major drivers that influenced the definition of the Ravenscar Profile are as follows:

- Inclusion of reliable and predictable building blocks for real-time systems.
- Elimination of non-deterministic and highly complex concurrency constructs.
- Support for a variety of application-level analytical verification models and techniques.
- Practical generation of formal evidence of safety and reliability certification for the implementation.

These drivers are highly complementary. The overall goal is twofold. First is the ability to develop application software that includes concurrency and interrupt-related activity in such a way that is suitable for analysis by sophisticated verification tools and techniques. Second is the ability to show early in the life cycle that the software implementation meets high integrity and safety-critical requirements.

The verification tools can provide evidence to the highest levels of assurance that the software meets the related requirements while also being free from run-time error. Examples of the kinds of verification tools and techniques that may be used with the profile include the following:

- Scheduling analyzers and response-time analyzers to show that all hard deadlines and data freshness require-

ments are met.
- Model checkers to show that the required system states exist and can be reached, and that no undesired states can occur.
- Static analyzers and formal proof tools to show that the code has been correctly constructed to meet its design specification and is free from run-time exceptions.

The motivation behind the execution environment to support implementations of the profile is to satisfy the following real-time and high integrity constraints: the footprint is small; the scheduling, synchronization, and timing characteristics are deterministic; the timing accuracy is at the resolution of the underlying system clock; and the run-time support library is simple enough to generate evidence of predictability, reliability, and safety.

Definition

The Ravenscar Profile is formally defined in terms of Ada95 constructs, and this definition has been accepted for inclusion in the revision to the ISO standard definition of the Ada language that is scheduled for 2005 release. The full definition is contained in a guide on using the Ravenscar Profile for high integrity systems [3]. The main components of a Ravenscar program are as follows:

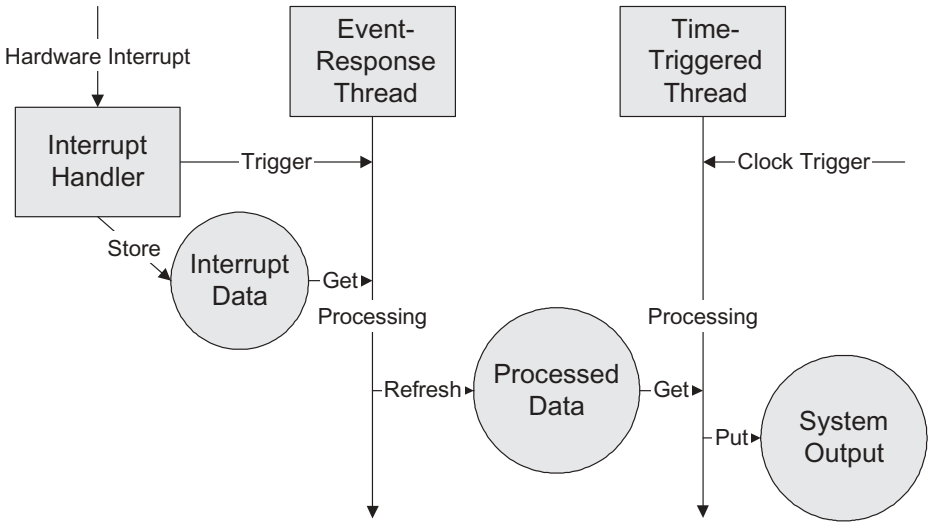
1. A fixed set of threads that may be cyclic (time-triggered) or aperiodic (event-triggered), including a thread parameterization mechanism.
2. A fixed set of protected objects that provide mutually exclusive access to shared data, including a protected object parameterization mechanism.
3. A fixed set of synchronization objects that provide suspend/resume capability for threads, including the communication of protected data as part of resumption.
4. A fixed set of interrupt handlers that may store data under mutually exclusive protection.
5. A synchronous delay facility based on absolute time values that are accurate to the resolution of the underlying system clock.
6. A deterministic fixed-priority preemptive thread scheduling policy.
7. A policy to enforce mutual exclusion that prevents deadlocks and minimizes the worst-case time that a thread is blocked due to contention.

Figure 1 illustrates the combination of some of these components. In this small example, a hardware interrupt is delivered when fresh external data is available to the system. The interrupt handler stores the data and triggers a response thread to process it. The processed data is stored in a shared data store, and a cyclic thread periodically obtains the latest version of this data to further process it to drive a system output.

Verification

Each thread of control is independently verified for conformance to its specification. This includes a demonstration of meeting its functional, performance, and resource utilization requirements, for example, by performing requirements-based testing or by using static analysis methods. Then the program as a whole is verified against all of its concurrency requirements, which include the following:
- Synchronization and communication interactions.
- Freshness of shared data.
- Execution order dependencies.
- Timing constraints such as meeting deadlines.

Figure 1: *Examples of Ravenscar Profile Building Blocks in Combination*



In each of these cases, sophisticated tools and techniques exist to automate the verification process to show that concurrency requirements have been met and to produce supporting evidence for a regulatory authority if necessary. The tool-based approach can be used early in the development life cycle and also simplifies the process of re-verification (and perhaps re-certification) after the system has undergone modification during maintenance or a midlife update.

In the rest of this section we look at three currently supported techniques for concurrency verification:
1. Static analysis.
2. Scheduling analysis.
3. Formal analysis.

Static Analysis

Existing static analysis tools and techniques can be used to achieve high levels of proof of correctness and absence of run-time errors in sequential programs, for example, see [4, 5]. The SPARK language recently has been extended to support the Ravenscar Profile as its concurrency model in such a way as to preserve the same level of integrity assurance as is possible for sequential programs [6]. This is a major advance in the extent of achievable confidence that concurrent programs are provably correct and cannot result in run-time exceptions being raised.

At the thread level, relating to an individual task or interrupt handler, the analysis is largely unaffected by the addition of concurrency constructs. In particular, the static analysis does not consider the temporal aspects – for example, the thread-level data and information flow analysis assumes that the thread will be activated after suspension at some stage.

The main change to existing sequential flow analysis is that references to shareable, protected objects must be considered volatile at all times because the value read may be generated by another program thread at any time. In particular, if a thread writes a shareable, protected object and later reads it, there can be no assumption that the value written will still be there when the read is performed. This volatility is already supported for sequential programs that access external data such as via an input/output port. Having modeled the volatility of shared data in this way, the existing benefits of proof of correctness and absence of run-time errors can be realized for each thread.

At the program level, the major extension to static analysis to support concurrency is to be able to describe the intended data and information flow across

thread boundaries, and then to verify that the actual program achieves it. The check is realized by the composition of each thread-level flow analysis with those of the thread interactions via shareable, protected objects. The intended program-wide flow relation can then be compared automatically with the computed actual flow, and any discrepancies reported as errors.

A valuable side effect of this analysis is in the assurance that the constructed program is complete. If a thread were inadvertently omitted from the program build, the computed flow analysis would not take into account the data interactions caused by that thread – hence the computed flow analysis would report an error when compared with the intended flows.

Scheduling Analysis

Recent research in scheduling theory has found that accurate analysis of real-time behavior is possible given a careful choice of the scheduling/dispatching method together with suitable restrictions on the interactions allowed between threads, for example, see [7] chapter 13. An example of a scheduling method is *preemptive fixed priority scheduling*. Example analysis schemes are *Rate Monotonic Analysis* (RMA) and *Response Time Analysis* (RTA). Preemptive fixed priority scheduling is generally used with a deterministic mutual exclusion policy such as *Priority Ceiling Protocol* (PCP) to avoid unbounded priority inversion and deadlocks. This provides a model suitable for the scheduling analysis of concurrent real-time systems that is also scaleable to programs for distributed systems.

This model supports *cyclic* and *aperiodic* threads that communicate and synchronize in a controlled way, and that each may have timing deadlines. These deadlines may be the following:

- **Hard.** When the failure to meet the deadline is an unacceptable failure of the system.
- **Firm.** When occasional missed deadlines can be tolerated but there is no value in completing the action when the deadline is missed.
- **Soft.** When occasional missed deadlines can be tolerated and there is value in completing the action when the deadline is missed.

The Ravenscar Profile requires using the standard preemptive fixed priority thread scheduling policy known as *First-In-First-Out* (FIFO)_Within_Priorities, and using PCP.

Extensive and mature tool support exists for both RMA and RTA, and for the static simulation of concurrent real-time programs. The primary aim of analyzing the real-time behavior of a system is to determine whether it can be scheduled in such a way that it is guaranteed to meet its timing constraints. Whether the timing constraints are appropriate for meeting the requirements of the system is not an issue for scheduling analysis. Such verification requires a more formal model of the program and the application of techniques such as model checking (see below).

Formal Analysis

The formal analysis of concurrent programs has been a fruitful research topic for a number of years. Current standard techniques allow many important properties of a concurrent program to be statically checked, for example the following:

- **Dependability.** The set of threads should not enter any undesirable state (for example deadlock, livelock).
- **Liveness.** All desirable states of the set of threads must be reached eventually (that is, useful progress should always be made).

In a real-time concurrent system, liveness becomes *bounded liveness* since desirable states must be reached by known deadlines.

Standard programming languages do not have their semantics defined in a formal mathematical way. Hence it is necessary to link a model of the program to the program itself. This link cannot be formal, but can be precise. Using standard patterns as found in the Ravenscar Profile helps this linkage. The formal model could be derived from the code or, more likely in an engineering process, the model is derived from requirements, and the code is obtained via a series of refinements from the model.

Verification is via either a proof-of-theoretic approach or model checking. An algebraic description can be proved to be deadlock free, for example, by using a theorem prover. Alternatively, a state transition description can be exercised by an exhaustive search of the set of states the program can enter. This *checking of the model* can deduce that all desirable states, and no undesirable states, can be reached.

For real-time systems, it is possible to add time parameters to the concurrency model and to then validate temporal aspects of the program. A common formalism for this type of state transition system is called a timed automaton. Tool support for model checking sets of timed automata is well advanced. One of the very useful features of model checking tools is that they all produce a well-defined counter example for any failed check.

There is already experience using model checking to validate Ravenscar programs. It is possible to add worst-case and best-case execution times for state transitions and to then check that deadlines are never missed. Alternatively, model checking can be used to validate the top-level description of the timing constraints, leaving scheduling analysis to check deadline satisfaction once execution times from the implementation are known. Typical of the verification that can be achieved with this approach is to check some end-to-end deadline through a number of threads, assuming that each thread itself meets its timing requirements.

Implementations

There are several mature implementations of the Ravenscar Profile in Ada95. These include Ada run-time systems that execute directly on the target board, and those that rely on services provided by a commercial off-the-shelf (COTS) real-time operating system (RTOS). Some of these implementations are part of systems that have achieved formal safety certification to the highest integrity level, for example Radio Technical Commission for Aeronautics, Inc. (RTCA)/ Defense Order (DO)-178B [2] Level A. In addition, there is growing support for the profile's building blocks within COTS RTOSs in a high-level language-neutral manner such that C programs can use them, for example.

Example

We can apply concurrency verification techniques to the simple example in Figure 1. By assigning deadlines to both threads, model checking would be able to verify that data from the interrupt was sufficiently fresh when used to influence the system output. Moreover once the execution times were known for the two threads (and the interrupt handler), it would be possible to use scheduling analysis to confirm that the deadlines for each thread would indeed be met in all possible executions of the program. Finally, static analysis could show correct system-wide data and information flow, for example, that the occurrence of the interrupt directly affects the system output, as well as absence of run-time errors.

Figure 2 (see page 12) shows some Ada code fragments for the expression of the example in Figure 1 using Ravenscar Profile constructs.

Conclusion

The Ravenscar Profile offers a unique

```
protected Interrupt_Data is
   procedure Handler; -- The interrupt handler code
   pragma Attach_Handler (Handler, Interrupt);
   entry Get (New_Data : out Raw_Data); -- Retrieves the interrupt data
private
   The_Interrupt_Data : Raw_Data;
end Interrupt_Data;

protected Processed_Data is
   procedure Refresh (New_Data : Data); -- Updates the processed data
   procedure Get (Latest_Data : out Data); -- Gets the latest data
private
   The_Processed_Data : Data;
end Processed_Data;

task body Event_Response is
begin
   loop
      Interrupt_Data.Get (New_Data); -- Waits until new interrupt data is available
      Process (New_Data, Output_Data); -- Processes it
      Processed_Data.Refresh (Output_Data); -- Writes the new processed data
   end loop;
end Event_Response;

task body Time_Triggered is
begin
   loop
      delay until Next_Period; -- Waits until start of next cycle
      Processed_Data.Get (Latest_Data); -- Gets the latest processed data
      Process (Latest_Data, New_Output); -- Processes it further
      System_Output.Write (New_Output); -- Writes the new system output
   end loop;
end Time_Triggered;
```

Figure 2: *Ada Code Fragments of Ravenscar Profile Constructs*

opportunity to developers of real-time and high integrity systems to establish high levels of confidence in the verification of concurrency properties and requirements early in the development life cycle. The profile defines a set of building blocks for constructing deterministic, concurrent software. The benefits of using the Ravenscar Profile include portability via international standardization, plus support from a wide range of sophisticated analysis tools. In addition, there exist implementations of the profile to the highest levels of safety certification. As a result, there is the opportunity to minimize the risk of deploying complex concurrent systems containing errors that are hard to find by testing methods alone, both during initial production and during long-term maintenance.◆

References

1. *Ada95 Reference Manual*. Cambridge, MA: Intermetrics, 1995. International Standard ISO/IEC 8652:1995(E) <www.adahome.com/rm95> and <www.adapower.com/rm95/index.html>.
2. RTCA-EUROCAE. Software Considerations in Airborne Systems and Equipment Certification. DO-178B/ED-12B. Dec. 1992 <www.rtca.org>.
3. Burns, Alan, Brian Dobbing, and Tuillo Vardanega. "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems." York, United Kingdom: University of York, Jan. 2003. Technical Report YCS 348 <www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf>.
4. Barnes, John. High Integrity Software – The SPARK Approach to Safety and Security. Addison-Wesley, 25 Apr. 2003.
5. Chapman, Rod, and Peter Amey. Industrial Strength Exception Freedom. Proc. of ACM SIGAda, Houston, TX, 2002 <www.sparkada.com>.
6. Amey, Peter, and Brian Dobbing. High Integrity Ravenscar. Proc. of Reliable Software Technologies – Ada-Europe 2003, Toulouse, France, June 2003. <www.sparkada.com/downloads/high_ integrity_ravenscar. pdf>.
7. Wellings, Andrew J., and Alan Burns. Real-Time Systems and Programming Languages. 3rd ed. Addison-Wesley, 5 Apr. 2001.

About the Authors

Brian Dobbing is a principal consultant at Praxis Critical Systems. He was a key member of the International Real-Time Ada Workshop that defined the first version of the Ravenscar Profile, and has been heavily involved in the evolution of the profile ever since. He was the chief architect of the first implementation of the profile, the Aonix product ObjectAda/Raven, that achieved formal safety certification to RTCA DO-178B Level A. He is a member of ISO Working Group 9 (Ada) and of the ISO Annex H Rapporteur Group (high integrity systems).

Praxis Critical Systems Ltd
20 Manvers St.
BATH BA1 1PX
U.K.
Phone: +44 1225 823762
E-mail: brian.dobbing@praxis-cs.co.uk

Alan Burns is head of the Computer Science Department, personal chair, and professor at the University of York, which he joined in January 1990. He has worked for many years on a number of different aspects of real-time systems engineering. His research activities have covered all major aspects of real-time and safety critical systems. Burns recently retired as chair of the Institute of Electrical and Electronics Engineers' Technical Committee on Real Time and has chaired the Real-Time Systems Symposium. He has authored/co-authored more than 350 papers and reports and eight books. His teaching activities include courses in Operating Systems, Scheduling, and Real-Time Systems.

Department of Computer Science
University of York
Heslington
YO105DD
York, U.K.
Phone: +44 1904 432779
E-mail: alan.burns@cs.york.ac.uk